Java Sound, Getting Started, Part 1, Playback

Baldwin shows you how to use the Sound API to play back previously captured audio data.

Published: January 21, 2003
By Richard G. Baldwin

Java Programming Notes # 2008

- Preface
- Preview
- Discussion and Sample Code
- Run the Program
- Summary
- What's Next?
- Complete Program Listing

Preface

This is the second lesson in a series of lessons designed to teach you how to use the Java Sound API. The first lesson in the series was entitled Java Sound, An Introduction.

What is sound?

From a human perspective, sound is the sensation that we experience when pressure waves impinge upon the small parts contained inside our ears.

The ultimate purpose of the Sound API is to assist you in writing programs that will cause specific sound pressure waves to impinge upon the ears of targeted individuals at specific times.

Two types of sound

Two significantly different types of audio *(or sound)* data are supported by the Java Sound API:

- Sampled audio data
- Musical Instrument Digital Interface (MIDI) data

I explained the difference between the two types of audio data in the previous lesson. Because the two types of audio data are so different, I am concentrating on sampled audio data at this point in time. I will defer any detailed discussion of MIDI data until later.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

The Java Sound API is based on the concept of *lines* and *mixers.*

I will provide a description of the physical and electrical characteristics of sound, in preparation for the introduction of an *audio mixer.*

I will use the scenario of a rock concert with six microphones and two stereo speakers to describe one of the ways that audio mixers are used.

I will discuss a variety of Java Sound programming topics, including mixers, lines, data format, etc.

I will explain the general relationship that exists among **SourceDataLine** objects, **Clip** objects, **Mixer** objects, **AudioFormat** objects, and ports in a simple audio output program.

I will provide a program that you can use to first capture and then to play back audio sound.

I will provide a detailed explanation of the code used to play back the audio data captured in memory by this program.

I will defer a detailed explanation of the code used to capture the audio data until the next lesson in the series.

Discussion and Sample Code

Physical and electrical characteristics of sound

The purpose of this lesson is to get you started writing programs using the Java Sound API.

The Sound API is based on the concept of an audio *mixer,* which is a device commonly used in the production of sound for concerts, music CDs, etc. Before getting into the details of an audio

mixer, it will be useful to review the physical and electrical characteristics of sound. Consider the picture shown in Figure 1.

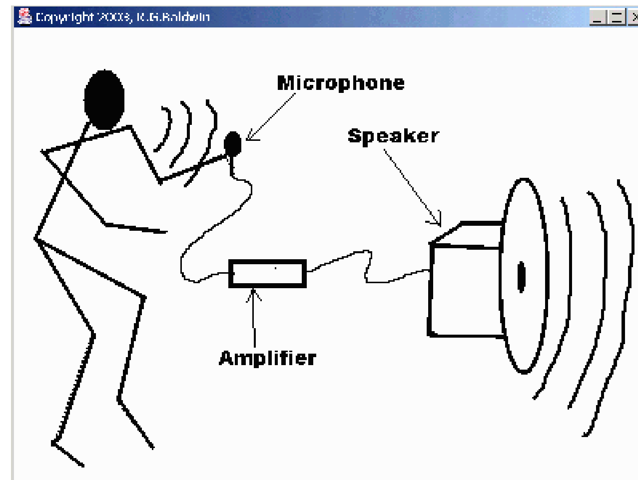

Figure 1 Joe Politico making a speech

Joe Politico making a speech

Figure 1 shows a picture of a politician named Joe Politico, making a speech using a system commonly known as a PA or *public address* system. A PA system typically consists of a microphone, an amplifier, and a loud speaker. The purpose of the PA system is to amplify Joe's voice so that it can be heard in the back of the crowd.

Vibrations in the air

Briefly, when Joe speaks, his vocal cords cause the air particles in his throat to vibrate. This, in turn causes sound pressure waves to impinge on a microphone, where the vibrations are converted to very low level electrical waves that mimic the sound pressure waves. The electrical waves are fed into an amplifier that amplifies the electrical waves. From there, the electrical waves are fed into a loud speaker, which converts the amplified electrical waves back into high-intensity sound pressure waves that mimic the original sound pressure waves created by Joe's vocal cords.

A dynamic microphone

Now consider Figure 2, which shows a schematic diagram of a type of microphone known as a dynamic microphone.

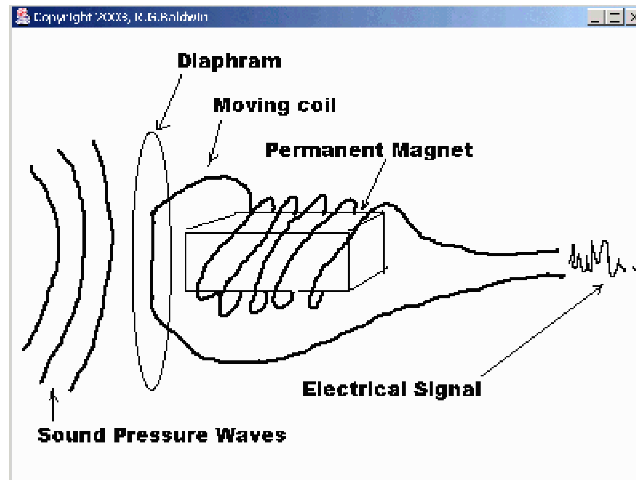

Figure 2 Schematic of a dynamic microphone

Sound pressure waves impinge on a diaphragm

Sound pressure waves created by Joe's vocal cords impinge on a flexible diaphragm inside the microphone. The sound pressure waves cause the diaphragm to vibrate, and the vibrations of the diaphragm mimic the sound pressure waves.

A coil of wire is caused to move

A coil made of very fine wire is attached to the diaphragm. As the diaphragm vibrates, the coil is caused to move back and forth in a magnetic field caused by a strong permanent magnet. It is a well-known fact that when a coil of wire cuts through a magnetic field, a voltage is induced in the coil.

> *(In fact, your local electrical power station generates power by spinning a huge coil in a very strong magnetic field. This results in the voltage that you use to drive the appliances in your house.)*

Electrical signal mimics Joe's sound pressure waves

Thus, a very weak voltage is induced in the coil in the microphone, and the waveform of the voltage mimics the sound pressure waves that impinge on the diaphragm. This is the voltage that is fed into the amplifier in Figure 1.

A loud speaker

As it turns out, the principle behind a loud speaker is simply a dynamic microphone operated in the reverse direction, as shown in Figure 3. *(Obviously, the wires are bigger and the diaphragm is larger in order to handle the amplified signals.)*

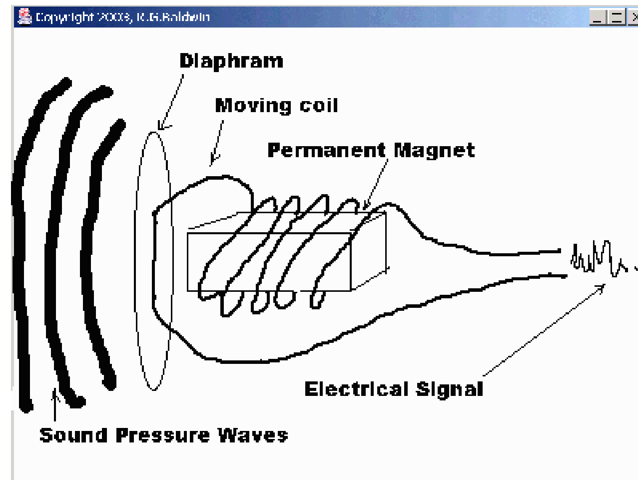

Figure 3 Schematic diagram of a loud speaker

In Figure 3, the amplified electrical signal, whose waveform mimics Joe's sound pressure waves, is applied to the moving coil on the right side of the figure. This moving coil floats in a strong magnetic field and is attached to a large flexible diaphragm.

Another well-known fact

It is a well-know fact that when you pass a current through a coil of wire, a magnetic field is induced. The interaction of the magnetic field produced by the coil and the magnetic field created by the permanent magnet causes the coil to move back and forth. This causes the diaphragm to vibrate.

Another vibrating diaphragm

The vibrations of the diaphragm in the loud speaker cause the surrounding air particles to vibrate and to create high-intensity sound pressure waves. The waveform of these high-intensity sound pressure waves mimic the low-intensity sound pressure waves created by Joe's vocal cords. The new sound pressure waves are strong enough to impinge upon the ears of the persons in the back of the crowd. Thus, Joe's amplified voice can be heard in the back of the crowd.

A rock concert

By now, you might be asking what this has to do with the Java Sound API. This is my lead-in to the concept of an audio mixer.

The scenario discussed above is fairly simple. It consists of one set of vocal cords, one microphone, one amplifier, and one loud speaker. Now consider the scenario depicted in Figure 4, which shows the stage being prepared for a rock concert.

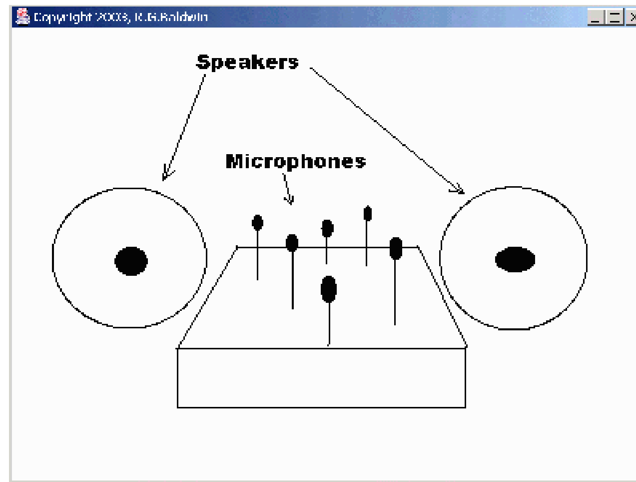

Figure 4 The stage for a rock concert

Six microphones and two speakers

In Figure 4, six separate microphones have been placed on the stage. Two loud speakers are placed on either side of the stage. When the concert begins, different performers will sing and play music into each of the six microphones. Each of the microphones will produce electrical signals, which must be individually amplified and fed into the two loud speakers. In addition, special sound effects, such as reverberation may be applied to the electrical signals before they are fed into the loud speakers.

A stereo experience

The two loud speakers are intended to provide a stereo audio experience. For example, all of the electrical signal produced by the right-most microphone might be applied only to the speaker on the right. Similarly, all of the electrical signal produced by the left-most microphone might be applied only to the speaker on the left. The electrical signals produced by the other microphones might be applied to the two speakers in proportion to their positions from left to right between the speakers. The electrical signals from the two microphones in the center might be applied equally to the two speakers.

An audio mixer

This task is often accomplished with an electronic device referred to as an *audio mixer.*

> *(If you enter the keywords **audio mixer concert -mp3** into the Google search engine, you can view material on hundreds of actual audio mixers. As of the date of this writing, you can read the specifications for, and view a picture of one such audio mixer (chosen at random) at the following URL.)*

An audio line

Although I am no expert on audio mixers, it is my understanding that a typical mixer has the ability to accept many independent electrical signals, each of which represents an independent audio signal or *line*.

> *(The concept of an audio **line** will become very important later when we get into the details of the Java Sound API.*
>
> *Perhaps this terminology derives from the fact that engineers and technicians often refer to the wires that carry electrical signals as lines. For example, I have two telephone lines in my house. We might also say that you should avoid high-voltage power lines. Also perhaps, the terminology derives from the fact that the controls for each audio signal in a mixer are often arranged in a line, as in the picture of the audio mixer referred to above.)*

Process each audio line independently

In any event, a typical audio mixer has the ability to apply amplification to each audio line independently of the amplification being applied to the other audio lines. The mixer may also have the ability to apply special audio effects, such as reverberation, to each of the lines. Finally, the mixer will have the ability to mix the individual amplified signals into output lines and to control the contribution that each input line makes to each output line. *(Controlling the contribution of each input line to each output line is often referred to as pan.)*

Back to the stereo experience

Thus, in the scenario pictured in Figure 4, the mixer operator has the ability to add together the signals produced by the six microphones in order to produce two output signals, each of which will be applied to one of the stereo loud speakers.

In effect, the signals from the six microphones can be added together to produce two output signals with the contribution of each microphone to each output signal being based on the physical position of the microphone on the stage. *(By changing the pan, a good mixer operator could even change that contribution over time as the lead singer moves back and forth across the stage.)* By controlling things in this way, the mixer operator can produce a stereo audio experience for the people in the audience.

Now to the programming world

Let's shift the discussion from the physical world to the programming world. According to Sun,

> *"Java Sound does not assume a specific audio hardware configuration; it is designed to allow different sorts of audio components to be installed on a system and accessed by the API. Java Sound supports common functionality such as input and output from a sound card (for example, for recording and playback of sound files)* ***as well as mixing of multiple streams of audio****."*

Mixers and Lines

Fundamentally, the sound API is constructed around the concept of *mixers* and *lines*. Moving from the physical world discussed above to the programming world, this is part of what Sun has to say about a mixer:

> *"A mixer is an audio device with one or more lines. It need not be designed for mixing audio signals. A mixer that actually mixes audio has multiple input (source) lines and at least one output (target) line.*
>
> *The former are often instances of classes that implement **SourceDataLine**, and the latter, **TargetDataLine**. **Port** objects, too, are either source lines or target lines. A mixer can accept prerecorded, loopable sound as input, by having some of its source lines be instances of objects that implement the **Clip** interface."*

The Line interface

Sun has this to say about the **Line** interface:

> *"A line is an element of the digital audio "pipeline," such as an audio input or output port, a mixer, or an audio data path into or out of a mixer. The audio data flowing through a line can be mono or multichannel (for example, stereo). ... A line can have **controls**, such as gain, pan, and reverb."*

Putting the terms together

The earlier quotation from Sun mentioned the following terms:

- SourceDataLine
- TargetDataLine
- Port
- Clip
- Controls

Figure 5 shows how four of these terms can come together to form a simple audio output program. *(I will discuss a simple audio input program in the next lesson.)*

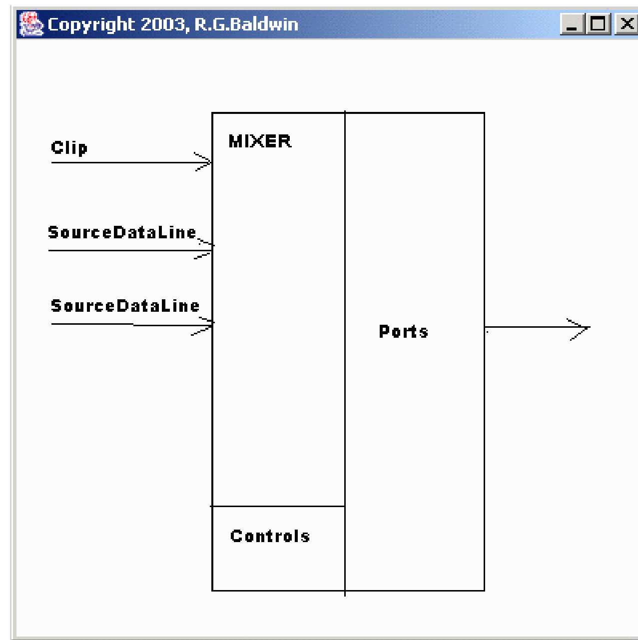

Figure 5 An audio output system

The programming scenario

From a programming viewpoint, Figure 5 indicates that a **Mixer** object has been obtained with one **Clip** object and two **SourceDataLine** objects.

What is a Clip?

A **Clip** is a mixer input object whose content doesn't change with time. In other words, you load the audio data into the **Clip** object prior to playback. The audio contents of the **Clip** can then be played back one or more times. You can cause the **Clip** to loop and continue playing the same audio data over and over.

A streaming input

A **SourceDataLine** object, on the other hand, is a *streaming* mixer input object. An object of this type can accept a stream of audio data and push that audio data into the mixer in real time. The actual audio data can derive from a variety of sources, such as an audio file, a network connection, or a buffer in memory.

Different kinds of lines

Thus, **Clip** objects and **SourceDataLine** objects can be viewed as input lines to a **Mixer** object. Each of these input lines can have its own *reverberation, gain,* and *pan* controls.

As indicated in Figure 5, the **Mixer** object can also include reverberation, gain, and pan controls.

Playing back audio

In this simple system, the **Mixer** reads data from the input lines, uses the controls to mix the input signals, and delivers the final output to one or more output ports, such as a speaker, a headphone jack, etc.

In the earlier lesson entitled *Java Sound, An Introduction,* I provided a simple program that captures audio data from a microphone port, stores that data in memory, and plays the captured data back through a speaker port. A copy of that program is provided in Listing 11 near the end of this lesson.

Will discuss capture and playback

A large portion of this program is dedicated to creating a graphical user interface that is used to control the operation of the program. I won't discuss those parts of the program. However, I will provide a discussion of the capture and playback portions of the program. I will discuss the playback portion in this lesson, and will discuss the capture portion in the next lesson. During this discussion, I will illustrate the use of audio lines with the Java Sound API.

Captured data stored in a ByteArrayOutputStream object

The capture portion of the program captures audio data from the microphone and stores it in a **ByteArrayOutputStream** object.

The playback method named **playAudio**, which begins in Listing 1, plays back the audio data that has been captured and saved in the **ByteArrayOutputStream** object.

```
  private void playAudio() {
    try{
      byte audioData[] =

byteArrayOutputStream.

toByteArray();

      InputStream byteArrayInputStream
            = new
ByteArrayInputStream(

audioData);
```

Listing 1

Begin with plain-vanilla code

The code fragment in Listing 1 really doesn't have anything in particular to do with Java Sound. Rather, this is plain-vanilla code whose purpose is to:

- Convert the previously saved data into an array of type **byte**.
- Get an input stream on the array of **byte** data.

This is necessary to make the audio data available for playback

Now for the Sound API.

The code in Listing 2 is very particular to the Java Sound API.

```
      AudioFormat audioFormat =

getAudioFormat();

Listing 2
```

At this point, I will briefly touch on a topic that I plan to explain in much more detail in a subsequent lesson.

Two independent formats

Frequently, there are two independent formats involved in audio data:

- The format of the file *(if any)* that contains the audio data *(there is no file involved in this program because the audio data is saved in memory).*
- The format of the sampled audio data itself.

What is the audio format?

Here is part of what Sun has to say about audio format:

> *"Each data line has an audio format associated with its data stream. The format (an instance of **AudioFormat**) specifies the arrangement of the bytes in the audio stream. Some of the format's properties are the number of channels, the sample rate, the sample size, and the encoding technique. Common encoding techniques include linear pulse-code modulation (PCM), mu-law encoding, and a-law encoding."*

A sequence of bytes

The sampled audio data consists of a sequence of bytes. There are a variety of optional ways that these bytes can be arranged and interpreted. I won't go into detail at this point regarding the many options. However, I will briefly discuss the format that is used for the audio data in this program.

A short side trip

At this point, I will depart from the method named **playAudio** and discuss the method named **getAudioFormat** *(that is called in Listing 2).* The entire method named **getAudioFormat** is shown in Listing 3.

```
  private AudioFormat
getAudioFormat(){
    float sampleRate = 8000.0F;
    int sampleSizeInBits = 16;
    int channels = 1;
    boolean signed = true;
    boolean bigEndian = false;
    return new AudioFormat(
                      sampleRate,

sampleSizeInBits,
                      channels,
                      signed,
                      bigEndian);
  }//end getAudioFormat

Listing 3
```

Aside from some initialized variable declarations, the code in Listing 3 consists of a single executable statement.

An AudioFormat object

The **getAudioFormat** method creates and returns an object of the **AudioFormat** class. Here is part of what Sun has to say about this class:

> *"**AudioFormat** is the class that specifies a particular arrangement of data in a sound stream. By examining the information stored in the audio format, you can discover how to interpret the bits in the binary sound data."*

Used the simplest constructor

The **AudioFormat** class has two constructors. *(I elected to use the simpler of the two.)* For this constructor, the required parameters are:

- Sample rate in samples per second. *(Allowable values include 8000, 11025, 16000, 22050, and 44100 samples per second.)*
- Sample size in bits. *(Allowable values include 8 and 16 bits per sample.)*
- Number of channels. *(Allowable values include 1 channel for mono and 2 channels for stereo.)*
- Signed or unsigned data. *(Allowable values include true and false for signed data or unsigned data.)*
- Big-endian or little-endian order. *(This has to do with the order in which the data bytes are stored in memory. You can learn about this topic here.)*

As you can see in Listing 3, this method specifies the following parameters for the new **AudioFormat** object:

- 8000 samples per second
- 16 bits per sample
- 1 channel *(mono)*
- Signed data
- Little-endian order

Default data encoding is linear PCM

The constructor that I used constructs an **AudioFormat** object with a linear PCM encoding and the parameters listed above *(I will have more to say about linear PCM encoding and other encoding schemes in future lessons).*

Now back to the playAudio method

Now that we understand something about the format of our audio data, let's return our attention to the method named **playAudio**. When we actually play the audio data later, we will need an object of the class named **AudioInputStream**. We instantiate such an object in Listing 4.

```
      audioInputStream =
        new AudioInputStream(
          byteArrayInputStream,
          audioFormat,

audioData.length/audioFormat.

getFrameSize());
```

Listing 4

Parameters for AudioInputStream constructor

The constructor for the **AudioInputStream** class requires the following three parameters:

- The stream on which this **AudioInputStream** object is based. *(As you can see, it is based on the* ***ByteArrayInputStream*** *object that was created earlier.)*
- The format of this stream's audio data. *(For this, we use the* ***AudioFormat*** *object created earlier.)*
- The length in sample frames of the data in this stream. *(See explanation below.)*

The first two parameters should be obvious from the code in Listing 4. However, the third parameter isn't quite as obvious.

Getting the length in sample frames

As you can see in Listing 4, the third parameter value is created by doing some arithmetic. One of the attributes of the audio format that I didn't mention earlier is something called a *frame*.

What is a frame?

For the simple linear PCM encoding scheme used in this program, a frame consists of the set of samples for all channels at a given point in time.

Therefore, the size of a frame *(in bytes)* is equal to the size of a sample *(in bytes)* multiplied by the number of channels.

As you have probably already guessed, the method named **getFrameSize** returns the frame size in bytes.

Calculating the length in sample frames

Therefore, the length of the audio data in sample frames can be calculated by dividing the total number of bytes in the audio data by the number of bytes in each sample frame. This is the calculation used for the third parameter in Listing 4.

Getting a SourceDataLine object

The portion of the program that we are discussing is a simple audio output system. As you may have surmised from the block diagram in Figure 5, we will need a **SourceDataLine** object to accomplish this task.

There are several ways to get a **SourceDataLine** object, none of which are particularly straightforward. The code in Listing 5 gets and saves a reference to an object of type **SourceDataLine**.

> *(Note that this code does not simply instantiate an object of the class* ***SourceDataLine****. Rather, it gets the object in a more roundabout fashion.)*

```
      DataLine.Info dataLineInfo =
                new DataLine.Info(

SourceDataLine.class,

audioFormat);

      sourceDataLine =
(SourceDataLine)

AudioSystem.getLine(

dataLineInfo);
```

Listing 5

What is a SourceDataLine object?

Here is part of what Sun has to say about the **SourceDataLine** type:

> *"A source data line is a data line to which data may be written. It acts as a source to its mixer. An application writes audio bytes to a source data line, which handles the buffering of the bytes and delivers them to the mixer. The mixer may ... deliver the mix to a target such as an output port ...*
>
> *Note that the naming convention for this interface reflects the relationship between the line and its mixer ..."*

The getLine method of the AudioSystem class

One of the ways of getting a **SourceDataLine** object is by invoking the static **getLine** method of the **AudioSystem** class *(I will have a lot more to say about the AudioSystem class in future lessons).*

The **getLine** method requires an incoming parameter of type **Line.Info**, and returns a **Line** object that matches the description in the specified **Line.Info** object.

Another short side trip

Sun has this to say about the **Line.Info** object:

> *"A line has an information object (an instance of Line.Info) that indicates what mixer (if any) sends its mixed audio data as output directly to the line, and what mixer (if any) gets audio data as input directly from the line. Subinterfaces of Line may have corresponding subclasses of Line.Info that provide other kinds of information specific to the particular types of line."*

A DataLine.Info object

The first statement in Listing 5 creates a new **DataLine.Info** object, which is a specialized *(subclass)* form of a **Line.Info** object.

There are several overloaded constructors for the **DataLine.Info** class. I elected to use the simplest one. This constructor requires two parameters.

A Class object

The first parameter is a **Class** object that represents the class of the data line described by the info object. As you can see, I specified this parameter as **SourceDataLine.class**.

The second parameter

The second parameter specifies the desired audio data format for the line. I provided the **AudioFormat** object created earlier for this parameter.

Are we there yet?

At this point, we still don't have the required **SourceDataLine** object. All we have so far is an object that provides information about the required **SourceDataLine** object.

Getting the SourceDataLine object

The second statement in Listing 5 creates and saves the required **SourceDataLine** object, by invoking the static **getLine** method of the **AudioSystem** class, and passing the info object as a parameter.

> *(In the next lesson, I will show you how to get such a **Line** object by operating directly on a **Mixer** object.)*

The **getLine** method returns a reference to the object as type **Line**, which is a superinterface of **SourceDataLine**. Therefore, a downcast is required before the return value can be saved as type **SourceDataLine**.

Preparing the SourceDataLine object for use

Once we have the **SourceDataLine** object, we must prepare it for use by *opening* and *starting* it, as shown in Listing 6.

```
sourceDataLine.open(audioFormat);
      sourceDataLine.start();

Listing 6
```

The open method

As you can see in Listing 6, I passed the **AudioFormat** object to the **open** method of the **SourceDataLine** object. According to Sun, this method:

> *"Opens the line with the specified format, causing the line to acquire any required system resources and become operational."*

Open status

Here is a little more of what Sun has to say on this topic:

> *"Opening and closing a line affects its resource allocation. Successful opening of a line guarantees that resources have been allocated to the line.*

Opening a mixer that has audio input and/or output ports normally involves acquiring the native platform hardware resource (sound card) and initializing any required software components.

Opening a line that is a data path in or out of the mixer might involve device initialization as well as allocation of limited resources from the mixer. In other words, a mixer has a finite number of lines, so at some point multiple applications (or the same application) might vie for usage of the mixer."

Invoking the start method on a line

According to Sun, invoking the **start** method on a line:

"Allows a line to engage in data I/O. If invoked on a line that is already running, this method does nothing. Unless the data in the buffer has been flushed, the line resumes I/O starting with the first frame that was unprocessed at the time the line was stopped."

In our case, of course, the line was never stopped. This was the first time that it was started.

Now we have most of what we need

At this point, we have all the audio resources that we need to play back the audio data that was previously captured and saved in the **ByteArrayOutputStream** object *(recall that this object exists only in memory).*

Going multithreaded

We will create and start a thread running to accomplish the actual playback. The code in Listing 7 creates the thread and starts it running.

*(Don't confuse the invocation of the start method on the thread with the invocation of the start method on the **SourceDataLine** object in Listing 6. Those are entirely different operations.)*

```
      Thread playThread =
          new Thread(new
PlayThread());
      playThread.start();
    } catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end playAudio

Listing 7
```

Straightforward code

The code in Listing 7 is straightforward, provided you understand multi-threaded programming in Java. If not, you can learn about multi-threaded programming by reviewing the tutorial lessons on the topic that I have published on my web site.

Once the thread has been started, it will run until all of the previously-captured data has been played back.

A new Thread object

The code in Listing 7 instantiates a new **Thread** object based on the class named **PlayThread**. The class named **PlayThread** is defined as an inner class in this program. The definition of the **PlayThread** class begins in Listing 8.

```
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];
```

Listing 8

The run method of the Thread class

Except for the declaration of an instance variable named **tempBuffer**, *(which is a **byte** array of size 10000 bytes),* the entire class definition is the definition of the required **run** method. As you already know, invoking the **start** method on a **Thread** object causes that object's **run** method to be executed.

The **run** method for this thread object begins in Listing 9.

```
  public void run(){
    try{
      int cnt;
      //Keep looping until the input
      // read method returns -1 for
      // empty stream.
      while((cnt = audioInputStream.
        read(tempBuffer, 0,
            tempBuffer.length)) != -
1){
        if(cnt > 0){
          //Write data to the internal
          // buffer of the data line
          // where it will be
delivered
          // to the speaker.
          sourceDataLine.write(
                   tempBuffer, 0,
cnt);
        }//end if
      }//end while
```

```
Listing 9
```

The first section of code in the run method

The **run** method consists of two major sections, the first of which is shown in its entirety in Listing 9.

In summary, a **while** loop is used to **read** audio data from the **AudioInputStream** object and to **write** that data to the **SourceDataLine** object.

Data written to the **SourceDataLine** object is automatically delivered to the speaker on the computer. A count value named **cnt** and a data buffer named **tempBuffer** are used to control the flow of data between the **read** and **write** operations.

The read method of AudioInputStream

The **read** method of the **AudioInputStream** reads up to a specified maximum number of bytes of data from the **AudioInputStream**, putting them into the specified byte array, beginning at the specified byte index, *(which in this case is zero).*

The return value

The read method returns the total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached. The number of bytes read into the buffer is saved in the variable named **cnt**.

The write method of SourceDataLine

If the number of bytes read is greater than zero, the **write** method of the **SourceDataLine** object is invoked. The **write** method writes audio data to the mixer via the source data line. The bytes are read from the specified array, starting at the given offset, and are written to the data line's buffer.

When the input stream is exhausted ...

When the **read** method returns -1, indicating that the input data is exhausted, control is transferred to the code in Listing 10.

```
      sourceDataLine.drain();
      sourceDataLine.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
```

```
}//end inner class PlayThread
```

Listing 10

Block and wait

The code in Listing 10 invokes the **drain** method on the **SourceDataLine** object to cause the program to block and wait for the internal buffer of the **SourceDataLine** to become empty. When it becomes empty, this means that all of the audio data has been delivered to the speaker.

Close the SourceDataLine

Then the code in Listing 10 invokes the **close** method to close the line, indicating that any system resources in use by the line can be released. Sun has this to say about closing a line:

> *"Closing a line indicates that any resources used by the line may now be released. To free up resources, applications should close lines whenever they are not in use, and must close all opened lines when exiting. Mixers are assumed to be shared system resources, and can be opened and closed repeatedly. Other lines may or may not support re-opening once they have been closed. Mechanisms for opening lines vary with the different sub-types and are documented where they are defined."*

End of story, for now

So there you have it, an explanation of how this program uses the Java Sound API to transfer audio data from an internal memory buffer to an external speaker.

Run the Program

At this point, you may find it useful to compile and run the program in Listing 11 near the end of the lesson.

Capture and playback audio data

This program demonstrates the ability to capture audio data from a microphone and to play it back through the speakers on your computer. The usage instructions are simple:

- Start the program running. The simple GUI shown in Figure 6 will appear on the screen.
- Click the **Capture** button and speak into the microphone.
- Click the **Stop** button to terminate capturing data.
- Click the **Playback** button to play your captured voice back through the system speakers.

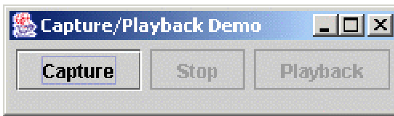

Figure 6 Program GUI

If you don't hear anything during playback, you may need to increase your speaker volume.

This program saves the data that it captures in memory, so be careful. If you attempt to save too much data, you may run out of memory.

Summary

I explained that the Java Sound API is based on the concept of *lines* and *mixers.*

I provided a description of the physical and electrical characteristics of sound, in preparation for the introduction of an *audio mixer.*

I used the scenario of a rock concert with six microphones and two stereo speakers to describe one of the ways that audio mixers are used.

I discussed a variety of Java Sound programming topics, including mixers, lines, data format, etc.

I explained the general relationship that exists among **SourceDataLine** objects, **Clip** objects, **Mixer** objects, **AudioFormat** objects, and ports in a simple audio output program.

I provided a program that you can use to first capture and then to play back audio sound.

I provided a detailed explanation of the code used to play back the audio data previously captured in memory by that program.

What's Next?

In this lesson, I explained that the Java Sound API is based on the concept of mixers and lines. However, the audio output code that I explained didn't obviously involve mixers. The **AudioSystem** class provides static methods that make it possible to write audio programs without having to deal directly with mixers. In other words, the static methods abstract mixers into the background.

In the next lesson, I will explain the audio capture code in a slightly modified version of the program that was discussed in this lesson. The modified version will make explicit use of mixers in order to show you how you can use them when you need to use them.

Complete Program Listing

A complete listing of the program is shown in Listing 11.

```
/*File AudioCapture01.java
This program demonstrates the capture
and subsequent playback of audio data.

A GUI appears on the screen containing
the following buttons:
Capture
Stop
Playback

Input data from a microphone is
captured and saved in a
ByteArrayOutputStream object when the
user clicks the Capture button.

Data capture stops when the user clicks
the Stop button.

Playback begins when the user clicks
the Playback button.

Tested using SDK 1.4.0 under Win2000
**************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.sound.sampled.*;

public class AudioCapture01
                        extends JFrame{

  boolean stopCapture = false;
  ByteArrayOutputStream
                 byteArrayOutputStream;
  AudioFormat audioFormat;
  TargetDataLine targetDataLine;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

  public static void main(
                        String args[]){
    new AudioCapture01();
  }//end main

  public AudioCapture01(){//constructor
    final JButton captureBtn =
                new JButton("Capture");
    final JButton stopBtn =
                   new JButton("Stop");
    final JButton playBtn =
               new JButton("Playback");
```

```
    captureBtn.setEnabled(true);
    stopBtn.setEnabled(false);
    playBtn.setEnabled(false);

    //Register anonymous listeners
    captureBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          captureBtn.setEnabled(false);
          stopBtn.setEnabled(true);
          playBtn.setEnabled(false);
          //Capture input data from the
          // microphone until the Stop
          // button is clicked.
          captureAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(captureBtn);

    stopBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          captureBtn.setEnabled(true);
          stopBtn.setEnabled(false);
          playBtn.setEnabled(true);
          //Terminate the capturing of
          // input data from the
          // microphone.
          stopCapture = true;
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(stopBtn);

    playBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          //Play back all of the data
          // that was saved during
          // capture.
          playAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(playBtn);

    getContentPane().setLayout(
                     new FlowLayout());
    setTitle("Capture/Playback Demo");
    setDefaultCloseOperation(
                        EXIT_ON_CLOSE);
    setSize(250,70);
```

```
    setVisible(true);
  }//end constructor

  //This method captures audio input
  // from a microphone and saves it in
  // a ByteArrayOutputStream object.
  private void captureAudio(){
    try{
      //Get everything set up for
      // capture
      audioFormat = getAudioFormat();
      DataLine.Info dataLineInfo =
                new DataLine.Info(
                  TargetDataLine.class,
                   audioFormat);
      targetDataLine = (TargetDataLine)
                   AudioSystem.getLine(
                         dataLineInfo);
      targetDataLine.open(audioFormat);
      targetDataLine.start();

      //Create a thread to capture the
      // microphone data and start it
      // running.  It will run until
      // the Stop button is clicked.
      Thread captureThread =
                new Thread(
                  new CaptureThread());
      captureThread.start();
    } catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end captureAudio method

  //This method plays back the audio
  // data that has been saved in the
  // ByteArrayOutputStream
  private void playAudio() {
    try{
      //Get everything set up for
      // playback.
      //Get the previously-saved data
      // into a byte array object.
      byte audioData[] =
                 byteArrayOutputStream.
                         toByteArray();
      //Get an input stream on the
      // byte array containing the data
      InputStream byteArrayInputStream
            = new ByteArrayInputStream(
                            audioData);
      AudioFormat audioFormat =
                      getAudioFormat();
      audioInputStream =
        new AudioInputStream(
```

```
          byteArrayInputStream,
          audioFormat,
          audioData.length/audioFormat.
                       getFrameSize());
      DataLine.Info dataLineInfo =
                new DataLine.Info(
                  SourceDataLine.class,
                          audioFormat);
      sourceDataLine = (SourceDataLine)
                   AudioSystem.getLine(
                         dataLineInfo);
      sourceDataLine.open(audioFormat);
      sourceDataLine.start();

      //Create a thread to play back
      // the data and start it
      // running.  It will run until
      // all the data has been played
      // back.
      Thread playThread =
          new Thread(new PlayThread());
      playThread.start();
    } catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end playAudio

  //This method creates and returns an
  // AudioFormat object for a given set
  // of format parameters.  If these
  // parameters don't work well for
  // you, try some of the other
  // allowable parameter values, which
  // are shown in comments following
  // the declarations.
  private AudioFormat getAudioFormat(){
    float sampleRate = 8000.0F;
    //8000,11025,16000,22050,44100
    int sampleSizeInBits = 16;
    //8,16
    int channels = 1;
    //1,2
    boolean signed = true;
    //true,false
    boolean bigEndian = false;
    //true,false
    return new AudioFormat(
                      sampleRate,
                      sampleSizeInBits,
                      channels,
                      signed,
                      bigEndian);
  }//end getAudioFormat
//===================================//
```

```
//Inner class to capture data from
// microphone
class CaptureThread extends Thread{
  //An arbitrary-size temporary holding
  // buffer
  byte tempBuffer[] = new byte[10000];
  public void run(){
    byteArrayOutputStream =
           new ByteArrayOutputStream();
    stopCapture = false;
    try{//Loop until stopCapture is set
        // by another thread that
        // services the Stop button.
      while(!stopCapture){
        //Read data from the internal
        // buffer of the data line.
        int cnt = targetDataLine.read(
                    tempBuffer,
                    0,
                    tempBuffer.length);
        if(cnt > 0){
          //Save data in output stream
          // object.
          byteArrayOutputStream.write(
                   tempBuffer, 0, cnt);
        }//end if
      }//end while
      byteArrayOutputStream.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
}//end inner class CaptureThread
//===================================//
//Inner class to play back the data
// that was saved.
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];

  public void run(){
    try{
      int cnt;
      //Keep looping until the input
      // read method returns -1 for
      // empty stream.
      while((cnt = audioInputStream.
        read(tempBuffer, 0,
            tempBuffer.length)) != -1){
        if(cnt > 0){
          //Write data to the internal
          // buffer of the data line
          // where it will be delivered
          // to the speaker.
          sourceDataLine.write(
                   tempBuffer, 0, cnt);
```

```
        }//end if
      }//end while
      //Block and wait for internal
      // buffer of the data line to
      // empty.
      sourceDataLine.drain();
      sourceDataLine.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
}//end inner class PlayThread
//===================================//

}//end outer class AudioCapture01.java
```

Listing 11

Copyright 2003, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

About the author

Richard Baldwin is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-